Universitat de Lleida

TREBALL FINAL DE GRAU

Estudiant: David Castillo Cabello

Titulació: Grau en Enginyeria Informàtica

Títol de Treball Final de Grau: Implementació d'un sistema per demostrar la possessió de dades

Director/a: Francesc Sebé Feixas

Presentació

Mes: Juliol

Any: 2018

Índex

Índex de figures **1**

1 Introducció **2**
1.1 Objectius 2
1.2 Metodología 3
1.3 Estructura del treball 3

2 Anàlisi d'un sistema PDP **4**
2.1 Anàlisi de Requeriments 4
2.2 Proposta de Solució 5
2.3 Anàlisi de seguretat 7

3 Disseny del sistema PDP **8**
3.1 Disseny de la fase de configuració 8
3.2 Disseny de la fase de verificació 11
3.3 Criptografia de clau simètrica 12

4 Implementació del sistema PDP **15**
4.1 Fase de configuració 15
4.2 Fase de verificació 19
4.3 Comunicació client-servidor 20

5 Resultats experimentals **22**
5.1 Estudi temporal 22
5.2 Estudi probabilitat 30

6 Conclusions **33**

7 Treball futur **34**

Referències **35**

Índex de figures

1 Pseudo-codi de la fase de configuració 5
2 Pseudo-codi de la fase de verificació 6
3 Diagrama de seqüència de la fase de configuració 9
4 Esquema d'una funció pseudo-aleatòria 9
5 Esquema d'una permutació pseudo-aleatòria 10
6 Esquema de l'algorisme AEk 11
7 Diagrama de seqüència de la fase de verificació 12
8 Gràfic temporal depenent de la mida del fitxer 27
9 Gràfic temporal depenent dels reptes que es volen realitzar . . . 28
10 Gràfic temporal depenent dels blocs per validació 29
11 Esquema on es detecta l'error 31
12 Esquema on no es detecta cap error 32

1 Introducció

En els últims anys, la necessitat d'emmagatzemar grans quantitats de dades ha crescut fins al punt que, en alguns casos, emmagatzemar les dades pròpies ha esdevingut inviable.
Per aquest motiu, han aparegut moltes empreses que ofereixen grans servidors amb una capacitat enorme, per emmagatzemar dades de tercers com a servei. Bàsicament, aquest servei consisteix en que el posseïdor de les dades confia en un servidor extern per emmagatzemar-les. Aquesta solució és bastant més econòmica i senzilla que si el client s'hagués d'encarregar de l'emmagatzematge i manteniment de les seves pròpies dades.
Alguns dels servidors més importants, a nivell global, són els de Amazon, Google o MEGA. En aquests casos, el servidor no solament emmagatzema, sinó que s'utilitza com a punt d'intercanvi de dades el que, en anglés, s'anomena *share point*. Un client deixa les seves dades al servidor, i aquestes es poden consultar en qualsevol moment, pel mateix client o per una tercera persona que hi tingui accés.
Aquest tipus de serveis generen una nova necessitat: com sabem que després de cinc anys sense accedir a aquelles dades, el servidor no les haurà esborrat per estalviar espai? Per afrontar aquesta necesitat han aparegut sistemes per provar la possessió de dades per part del servidor. Són tècniques per assegurar la integritat de les dades guardades en servidors externs.

1.1 Objectius

No és l'objectiu d'aquest projecte el construir un entorn client-servidor amb una estructura de bases de dades per emmagatzemar les dades. Tampoc intenta fer una implementació amb concurrència per tal que el servidor pugui emmagatzemar fitxers d'una mida exageradament gran, ja que no es disposa de la infraestructura suficient.
L'objectiu principal d'aquest treball és centrar-se en una d'aquestes tècniques. Concretament es vol dur a terme l'anàlisi i la implementació d'un sistema de prova de possessió de dades, en anglés, Provable Data Posesion, des d'ara PDP. El sistema estudiat està exposat a l'article científic d'Ateniese et al. "Scalable and Efficient Provable Data Possession"[4]. Aquest treball pretén implementar i estudiar l'algorisme proposat, conèixer l'eficiència del mateix i provar que re-

alment funciona com indiquen els seus autors.

1.2 Metodología

Primer de tot s'han estudiat les característiques de diversos sistemes PDP per conèixer l'objectiu i les tècniques que han aparegut. Una vegada realitzada aquesta investigació, el projecte es centra en el sistema proposat. El sistema escollit proposa un mètode per comprovar que realment el servidor és posseïdor de les dades del client.
Per assolir i poder fer més senzill aquest sistema, es proposa un disseny de la futura aplicació que contindrà l'algorisme que proposa el sistema proposat.
Una vegada estudiat, entès i dissenyat aquest sistema, es comença a implementar, sempre que es pugui amb les tècniques que els autors recomanen a l'article. En cas que no sigui viable utilitzar la tècnica proposada, s'implementa una altra forma eficient de fer-ho i equivalent a la proposada.
Quan ja s'ha dut a terme l'implementació es comença a fer l'estudi de rendiment, provant diferents casos el més propers possibles als que trobariem en una situació real.

1.3 Estructura del treball

El treball està format per tres parts fonamentals.
La primera, és l'anàlisi d'un sistema *Provable Data Possession*, on s'explica quins requeriments necessita el sistema per funcionar, quina solució hem acabat implementant i la seguretat que té aquesta.
Després, detallem la implementació del sistema, quines decisions de disseny s'han pres i perquè.
Per últim, el treball té un estudi a nivell de rendiment i temps d'execució que presenta la solució implementada, amb les conclusions que podem obtenir mitjançant la nostra versió dels algorismes proposats pel sistema PDP.

2 Anàlisi d'un sistema PDP

Com s'ha explicat a la introducció, amb la necessitat de conèixer si el servidor manté totes les dades del client, han aparegut molts sistemes per provar la possessió de dades(PDP), aquests sistemes, en ocasions, també s'han referenciat amb el nom de prova de recuperabilitat de dades, en anglés *Proof of Data Retrivability* (POR).

2.1 Anàlisi de Requeriments

L'objectiu principal del PDP és permetre al client verificar eficientment, frenqüentment i de manera segura que les seves dades continuen intactes i que el servidor no les ha esborrat ni modificat.
Amb una necessitat cada vegada major, han sorgit molts sistemes PDP proposats per diferents equips d'investigadors, per exemple, els articles presentats per G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson and D. Song, *"Provable data possession at untrusted stores"*[1] i A. Juels i B. Kaliski, *"PORs: Proofs of retriability for large files"*[2].

El primer, tracta de generar proves de possessió probabilístiques formades a partir de blocs aleatoris. El terme probabilístic s'està referint a que no són cent per cent fiables, però on la probabilitat de fallada és negligible.
Que les dades per la validació no es generin a partir del fitxer sencer, afavoreix a l'eficiència en la comunicació *client-server*. Aquest article també planteja el poder recalcular aquestes proves de possessió de dades quan el client modifiqui o afegeixi dades sense haver de repetir tots els càlculs des de l'inici.

El segon, introdueix la idea de generar funcions hash[3] per fer més eficient el coll d'ampolla resultat de la comunicació entre el client i el servidor.
Per poder validar que el servidor posseeix les dades, el client calcula un hash amb la seva clau secreta.
Per obtenir la prova de possessió, publica la clau secreta i reclama al servidor que calculi una resposta. El problema és que si el client vol més d'una prova de possessió, ha de disposar de més d'una clau, tantes claus com proves hagi de fer.

2.2 Proposta de Solució

Inspirat en els articles anteriors, va sorgir un altre algorisme, que tractava d'adaptar, l'eficiència del segon amb la forma dinàmica de calcular les claus del primer. Aquest article és el treball de G.Ateniese, R. Di Pietro, L. Mancini and G. Tsudik *"Scalable and Efficient Provable Data Possession"*[4], un sistema completament basat en criptografia de clau simètrica.

La principal idea d'aquest algorisme consisteix en que, abans d'emmagatzemar les dades, el client calcula un cert número de *tokens* de verificació. Cada un d'aquests verifica una part de les dades, dades que prèviament han estat dividides en blocs.
Una vegada calculat, el client envia aquests *tokens* juntament amb les dades, al servidor. Aquesta part és el primer algorisme del sistema, que a l'article ve donat amb el pseudo-codi que mostra la figura 1(pàgina 5).

Algorithm 1: Setup phase

begin
  Choose parameters c, l, k, L and functions f, g;
  Choose the number t of tokens;
  Choose the number r of indices per verification;
  Generate randomly master keys $W, Z, K \in \{0,1\}^k$.
  for $(i \leftarrow 1$ **to** $t)$ **do**
    begin *Round i*
1   Generate $k_i = f_W(i)$ and $c_i = f_Z(i)$
2   Compute $v_i = H(\, c_i \,,\, D[g_{k_i}(1)] \,,\, \ldots \,,\, D[g_{k_i}(r)] \,)$
3   Compute $v'_i = AE_K(\, i \,,\, v_i)$
    end
  Send to $\mathcal{SRV}$: $(D, \{[i, v'_i] \text{ for } 1 \leq i \leq t\})$
end

Figura 1: Pseudo-codi de la fase de configuració

A l'algorisme de la figura 1 els parametres són:

- f = pseudo random function, PRF
- g = pseudo random permutation, PRP
- c = mida de l'entrada a la PRF
- l = mida de la sortida de la PRP
- k = mida de les claus W,K,Z
- L = mida de la sortida de la PRF

Quan el client vol obtenir una prova de la possessió de dades del servidor, li demana que calculi un hash trobat a partir d'una clau aleatòria, creada a partir d'un índex que representa un bloc de les dades que ha de provar que encara posseeix. Desencriptant aquest hash amb una clau privada, el client, en cas de possessió de dades, obté un valor igual al *token* que el servidor li ha enviat. Aquest sistema de demanar una prova de possessió de dades és el segon algorisme del sistema, amb pseudo-codi mostrat a la figura 2(pàgina 6).

Algorithm 2: Verification phase

begin *Challenge i*

1 $\mathcal{OWN}$ computes $k_i = f_W(i)$ and $c_i = f_Z(i)$

2 $\mathcal{OWN}$ sends $\{k_i, c_i\}$ to $\mathcal{SRV}$

3 $\mathcal{SRV}$ computes $z = H(\, c_i \,,\, D[g_{k_i}(1)] \,,\, \ldots \,,\, D[g_{k_i}(r)] \,)$

4 $\mathcal{SRV}$ sends $\{z, v'_i\}$ to $\mathcal{OWN}$

5 $\mathcal{OWN}$ extracts v from v'_i. If decryption fails or $v \neq (i \,,\, z)$ then REJECT.

end

Figura 2: Pseudo-codi de la fase de verificació

Donat que les mides de les claus i els valors hash són relativament petites, aquest sistema és realment eficient en termes de computació i d'ample de banda.

2.3 Anàlisi de seguretat

Pel que fa a la seguretat, el sistema de PDP proposat per Ateniese et al.[4] també té un estudi de la seguretat d'aquest sistema.
En el seu article, els autors, expliquen a través d'un joc, versió d'un explicat en l'article *"Provable data possession at untrusted stores"*[1], que és altament improbable que el servidor, sense posseir les dades, doni una prova de possessió de dades correcta.
Segons els autors l'única forma de poder donar la prova de possessió correcta és trobar els valors representats pels hash, aquests valors es poden trobar a partir de la clau que ha donat el client al servidor, però això no és possible si les claus i els hash es generen de forma aleatòria. Per tant, si s'utilitza un valor hash generat a partir de una clau aleatoria, el client pot estar segur de que el servidor no pot falsejar les proves perquè és completament impossible obtenir les dades del hash que posseeix el servidor sense la clau privada del client.

Per altra banda, cal considerar la seguretat de la xarxa. Que passaria si un tercer intenta obtenir les dades del client en la comunicació amb el servidor? L'única forma de conèixer les dades del client mitjançant un atac per xarxa seria atacar la comunicació durant la fase de configuració que es fa solament una vegada. Aquest és el moment en que s'envien les dades sense xifrar. La resta de les comunicacions entre els dos participants, simplement, són cadenes de caràcters que representen valors hash. Mai es poden obtenir les dades originals del client a partir de la comunicació d'aquesta fase de verificació.

3 Disseny del sistema PDP

El sistema es divideix en dues fases molt difereciades.

3.1 Disseny de la fase de configuració

En aquesta primera fase en la que s'emmagatzemen les dades que es volen guardar i, juntament amb aquestes, el client escolleix una configuració depenent de les seves necessitats. Si necessita accedir a les dades ràpidament sobre l'opció d'accedir moltes vegades, preferirà una configuració on hi hagi un número més reduït de tokens i amb unes particions de blocs una mica més petites. Si per altra banda, vol poder reptar moltes vegades al servidor, escollirà molts tokens. Per últim, si vol un sistema molt més fiable, haurà d'escollir una configuració on les dades es divideixen molt més.
Per generar el sistema amb les diferents configuracions s'ha decidit generar una cadena de caràcters com a entrada del sistema, aquesta cadena te el següent format.

k=128,chalenges=n,r=m,d=n

On la nomenclatura és:

- k = mida de les claus aleatòries, actualment els unics valors disponibles són 128 o 256
- challenges = vegades que es reptarà al servidor
- r = blocs de dades utilitzat per crear les verificacions
- d = nombre de blocs en que es divideixen les dades
- n = número enter
- m = número enter entre 1 i n

Durant aquesta primera fase, el client fa una petició al sistema que s'encarrega de generar tres claus aleatòries. Amb les claus generades, comença un bucle amb tantes iteracions com el nombre de cops que el client vulgui reptar al servidor. Per cada una d'aquestes iteracions, el sistema genera els *tokens* que més endavant serviran per reptar al servidor en la fase de verificació.
Aquests *tokens* es generen a partir de dues PRF(*Pseudo-random function*, una

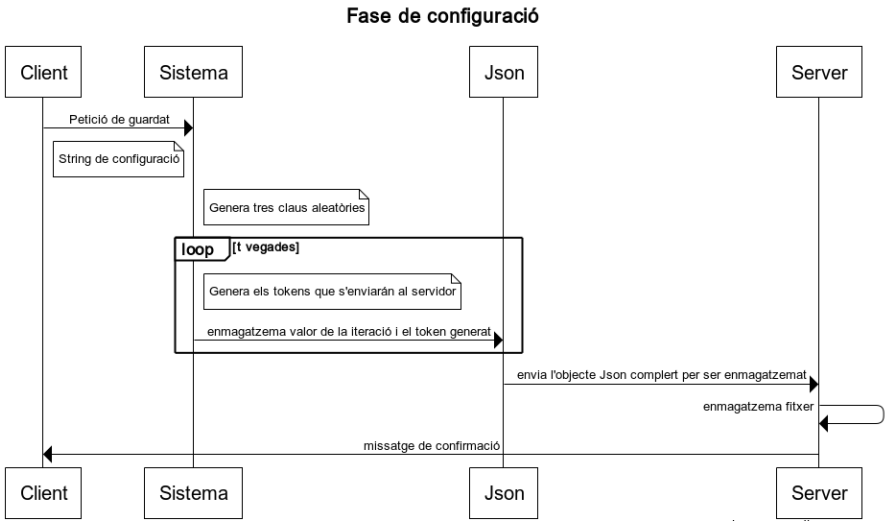

Figura 3: Diagrama de seqüència de la fase de configuració

funció pseudo-aleatòria). Així s'aconsegueix que el sistema no es pugui veure comprom és per un atac d'un tercer. Encara que les claus, les quals s'envien i treballaran durant tot el procés, estan generades a partir de les dades que el client envia, mai es podran obtenir per trobar informació parcial o total sobre aquestes dades, perquè són el resultat d'un hash generat a partir de dues claus trobades amb una PRF.

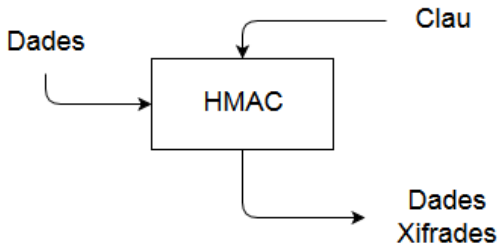

Figura 4: Esquema d'una funció pseudo-aleatòria

Una vegada trobades les dues noves claus, el sistema ha de generar el factor verificador, el cor del sistema, per això l'article proposa una permutació de les

dades que anomenen *pseudo-random permutation* amb les sigles PRP que al català es tradueix permutació pseudo-aleatòria, la qual s'encarrega de desordenar els blocs de dades mitjançant una de les claus privades anteriors. D'aquesta manera el servidor podrà reproduir el mateix *desordre* generat pel client.
Del resultat d'aquestes dades desordenades s'obté una cadena de caràcters, la qual es concatena amb una de les claus generades prèviament que no s'ha utiltzat per generar la permutació. Llavors, d'aquest resultat es genera un hash per tal de reduir la mida del resultat per a que la comunicació i comprovació siguin més eficients.

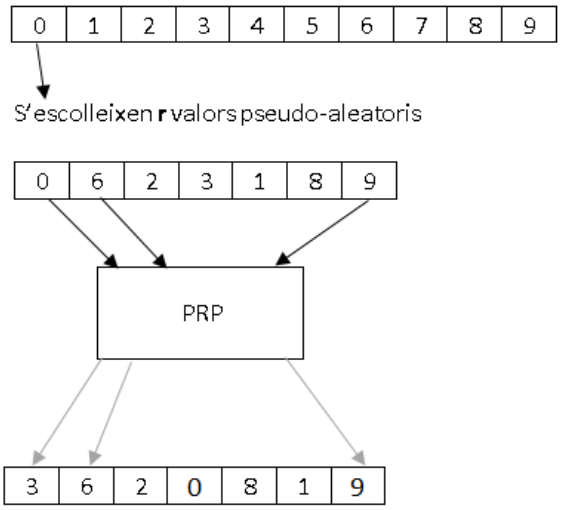

Figura 5: Esquema d'una permutació pseudo-aleatòria

Per últim, tenim l'algorisme que s'encarrega de verificar que el servidor manté les dades, el AEk. Es tracta d'un algorisme que assegura l'encriptació i l'autenticació del client. Es genera a partir del hash resultat del pas anterior, el qual ha de ser xifrat. En aquest cas s'ha escollit l'algorisme AES[5], i després el missatge encriptat ha de ser autenticat, en aquest cas, amb un codi HMAC[6]. El procediment és senzill. Primer les dades són encriptades i del resultat d'aquesta encriptació es genera una cadena de caràcters a la qual se li ha de concatenar el valor resultant de l'aplicació del HMAC sobre si mateixa.

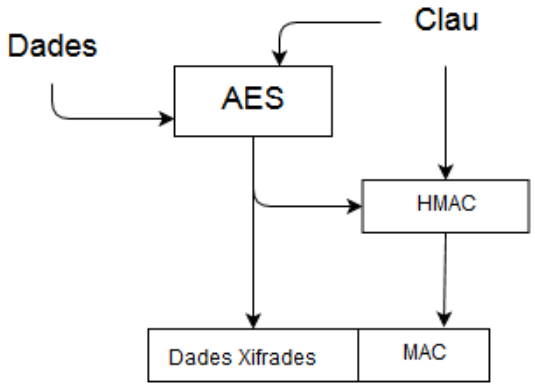

Figura 6: Esquema de l'algorisme AEk

3.2 Disseny de la fase de verificació

La segona fase s'encarrega de validar que el servidor encara posseeix les dades del client intactes. És la fase de verificació. Per tal de fer aquesta comprovació, el client repta al servidor demanant que li retorni la resposta al repte corresponent a cert índex.
En aquesta fase, el client del sistema té una càrrega de càlcul inferior a la del servidor.

El client envia la configuració que va guardar durant la fase prèvia, així com l'índex que haurà de validar el servidor i les claus que prèviament ha calculat amb l'algorisme PRF. Com es pot deduir, el client calcula aquestes claus cada vegada que ha de fer una petició al servidor.
Primer, al rebre les dades, el servidor accedeix dins d'aquestes, que estan en format JSON, concretament a la posició que conté l'índex demanat. D'aquí extreu tant l'índex com la clau que hi havia emmagatzemada (a partir d'ara li direm *token*).
A partir del moment en que el servidor té totes les dades, comença a calcular la clau que servirà per fer la validació. Per fer això, ha de fer els mateixos passos que va fer el client a la fase de configuració. És primordial que tant els blocs escollits aleatòriament del total de les dades com la permutació, siguin igual que les escollides anteriorment pel client, sinó la validació no funcionaria.
Generat el hash, el servidor envia al client tant la resposta del repte com el *token* que ha obtingut prèviament. Per validar les dades, el client ha d'agafar la

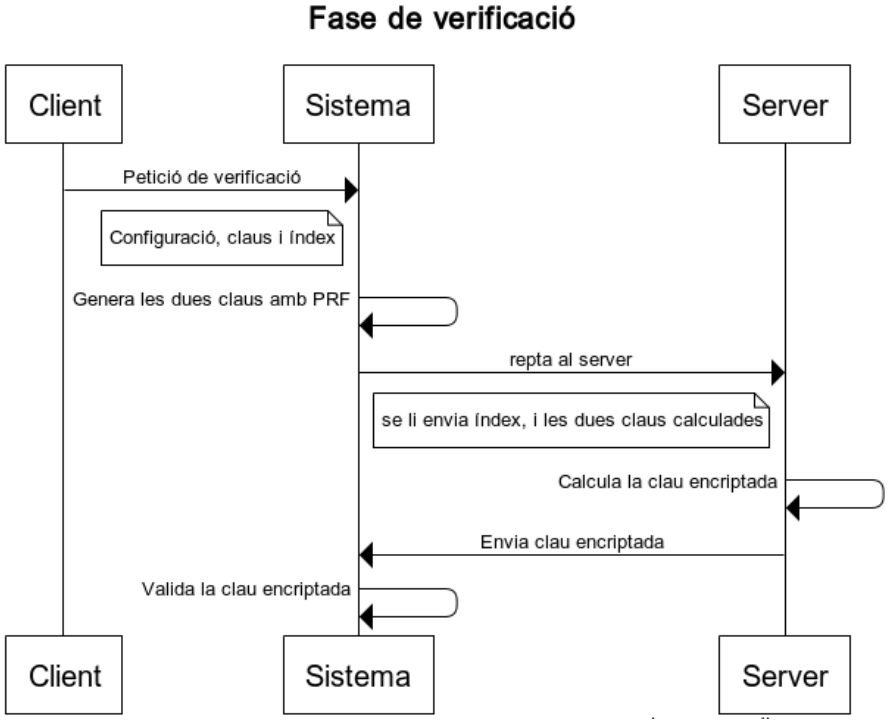

Figura 7: Diagrama de seqüència de la fase de verificació

clau de validació, aplicar-li l'algorisme AEk, com ja havia fet a la configuració, amb la seva clau privada, i si el resultat de l'algorisme és igual al *token* enviat pel servidor, el client pot asegurar que el servidor posseeix les seves dades.

3.3 Criptografia de clau simètrica

Durant el disseny apareix una de les preguntes més importants del sistema, per què aquest sistema és tan segur?

Per una banda, el servidor mai podrà conèixer les claus generades per la PRF, k_i i c_i, de la següent iteració ja que es generen a partir de les claus privades que ha calculat el client i solament aquest coneix, ja que mai han estat compartides.
Que el servidor no conegui aquestes claus implica que en cap moment, abans del repte, podrà precalcular la resposta correcta al repte proposat pel el client en el

futur. Això és possible gràcies a que, per a cada un d'aquests reptes, se li envia al servidor les claus que ha d'utilitzar per generar la clau que ha de validar. Amb aquest sistema, el client s'assegura que el servidor mai esborra o modifica les seves dades, ja que les ha de fer servir a cada un dels reptes proposats.
Aquest sistema es pot demostrar amb un senzill exemple. Imaginem que el client envia al servidor les seves dos primeres claus k_i i c_i. Per aquest exemple, la PRF que farem servir dóna el valor enter de la longitud de la clau privada dividida pel primer dígit i sumant el resultat del dígit situat a la posició de l'índex multiplicat pel mateix índex.
Les dades que solament posseeix el client són les seves claus:

- K = 12AB
- W = 2ABC34
- Z = 123DE

Segons la PRF, els valors de les claus compartides de la primera iteració són:

- c1 = fz(1) = 5/1 + 1x1 = 6
- k1 = fw(1) = 6/2 + 2x1 = 5

Per tant, el servidor tindrà constància dels valors de k_i i c_i per la primera iteració, pero no podrà obtenir les claus privades del client(K,W i Z). La única forma d'obtenir aquestes claus és per força bruta i hauria de posseir molts resultats d'iteracions anteriors per aconseguir-ho en aquest sistema tant senzill. Si fos un de més complex la quantitat d'iteracions creixeria exponencialment.

Per altra banda, el servidor podria provar d'obtenir les claus mitjançant les dades que té emmagatzemades.
Recordem que el servidor, juntament amb les dades té les claus *vi'*, que són el resultat d'aplicar l'algorisme AEk sobre cada una de les claus *vi*. Cada una de les claus *vi* és justament el que ha de calcular per validar la seva possessió de dades. Per tant, per trobar-ho tot, simplement hauria de calcular l'algorisme invers al AEk.
Com ho hauria de fer?
Primer hauria de posseir una de les claus privades que solament té el client,

la clau privada K. Amb la clau hauria de separar el missatge del MAC, que recordem que és el que valida l'autenticitat de les dades, calculat amb un hash i després, calcular la inversa del AES que s'utilitza per calcular el missatge que s'ha enviat.
Cosa gens senzilla, encara que la seguretat de la clau privada hagués estat compromesa, perquè el servidor, en cap moment, coneix quins algorismes s'han utilitzat per generar el hash ni la encriptació.
Com ja hem vist, la forma en que es calcula cada una de les claus *vi'* és la següent:
V_i' -> $AES_k(v_i)$ + $HMAC_k(AES_k(v_i))$
Per exemple, si el sistema disposa d'un vi' = ABC123DEF45 aquest no pot obtenir quins d'aquests caràcters formen part de l'algorisme d'autenticació i quins formen part de l'algorisme AES amb clau k.
Imaginem que aquests valors són els següents:

- AESk(vi) = ABC123D
- HMACk(AESk(vi)) = EF45

La forma de desencriptar el sistema AES és coneixent la clau privada del client, la clau k, ja que un dels fonaments d'aquest sistema, per a que sigui imposible de desencriptar, és guardar aquesta clau i mai compartir-la.
Per tant, encara coneixent quina part de *vi'* és del MAC i quina del AES no es pot conèixer la inversa d'aquesta funció, és a dir, no es poden precalcular totes les claus originals *vi* i esborrar les dades emmagatzemades.

4 Implementació del sistema PDP

Per dur a terme la implementació s'ha utilitzat el llenguatge de programació Python 2.7, ja que és un dels llenguatges més populars i amb més llibreries, tant pels algorismes de les funcions hash com per l'esquema de xifrat AES[10] que són les sigles de Advanced Encryption Standard. A més a més, Python permet programar sockets amb molta facilitat per crear una comunicació entre client i servidor amb una de les seves llibreries internes.
El sistema consta de dues fases diferenciades, com ja s'ha explicat a l'apartat de disseny, on s'executa una o l'altra depenent del que vulgui fer el client, emmagatzemar o reptar al servidor.
Primer, a l'hora d'emmagatzemar la informació, s'utilitza la fase de configuració o, en anglés, "set up phase", i després, per cada vegada que es vol reptar al servidor per comprovar la possessió de dades, s'utilitza la fase de verificació o repte, en anglès, 'Challenge phase'.

4.1 Fase de configuració

Les claus es generen en forma de cadena de caràcters, a la qual es van afegint els 0 o 1 escollits aleatòriament, amb la llibreria random de Python i, posteriorment, convertint el valor binari que representa aquesta cadena de dígits com un valor hexadecimal.

```
def randomBinaryKey(k):
    key = ""
    for i in range(k):
        if random.random() < 0.5:
            key = key+"0"
        else:
            key = key+"1"

    key = "%32X" % int(key,2)
    return key
```

Una vegada s'han generat les tres claus aleatòries **w**, **k** i **z**, s'han de produir tants tokens com vegades el client vulgui reptar al servidor. Cada token s'obté a partir d'un índex. Per aquest motiu, l'algorisme original proposa un bucle on

es van iterant números entre l'1 i el número de reptes que el client ha proposat a la configuració.

Per cada iteració s'ha de generar la clau **kx**, generada a partir de la clau **w**, i **cx**, generada a partir de la clau **z**, que són el resultat d'una funció pseudo-aleatòria. Per exemple, la generació de **kx** s'ha implementat de la manera següent:

```
kx = hmac.new(keys.w)
kx.update(str(x))
kx = kx.hexdigest()
```

La variable **kx** pren el valor obtingut d'una funció hash basada en la llibreria HMAC que ofereix Python. HMAC prové de l'anglés "keyed-hash message authentication code", que és un tipus de un codi de autenticació de missatge. Les seves sigles amb anglés són MAC. En aquest cas la funció està basada en l'algorisme MD5[8], l'algorisme estàndard, el qual retorna un resultat amb una mida de 128 bits, adequat a les mides amb les que es necesita treballar en aquesta part de l'algorisme d'elaboració de claus. A la funció HMAC se li ha d'enviar una clau per generar el valor hash. En aquest cas, se li pasa la clau privada W i el que es pretén és obtenir un valor hash del missatge x que representa l'índex que s'està iterant en aquesta volta del bucle.
Com es pot veure al codi, a la variable *kx* se li aplica una funció *hexdigest*. El motiu és que *kx* ha estat creat com un objecte de tipus *hmac*. El que s'aconsegueix amb la funció *hexdigest* és convertir aquest objecte de tipus *hmac* en una cadena de caràcters, per poder treballar amb aquest codi hash en el futur amb molta més facilitat, poder transformar-lo i concatenar-lo.

Del procés anterior s'obtenen les dues claus resultat de les funcions hash, *cx* i *kx*. La clau *cx* formarà part de la validació que ha de fer el client en el moment en que el servidor li envii les dades per ser comprovades. La segona part de la validació ve donada pels blocs de dades escollits aleatòriament de manera desordenada, amb una permutació *Pseudo-Random Permutation*, a partir d'ara PRP, generada a partir de la clau *kx*.

```
def permutation_iter(r, kx,nB):
    aux = []
    random.seed(kx)
```

```
    for it in xrange(r):
        aux.append(random.randint(0,nB-1))

    for elem in xrange(r):
        rand = random.randint(0,r-1)
        tmp = aux[elem]
        aux[elem] = aux[rand]
        aux[rand] = tmp

    return aux
```

La PRP s'ha implementat com es pot veure al codi anterior. Primer, es genera una llista d'índexs entre 0 i el numero de tokens que ha escollit l'usuari per fer les validacions. Després amb la llavor, la clau *kx*, es generen els números aleatoris. Aquests s'intercanvien amb el nombre que té l'índex aleatori que s'ha generat i així es genera un array d'índexs que representen la posició dels blocs de dades. Així s'obté una cadena de caràcters de les dades desordenades.

```
vx = hashlib.sha256()
vx.update(inputKey)
vx = vx.hexdigest()
```

Sobre el resultat de la concatenació de *cx* i els blocs de dades desordenats, resultat del PRP, s'ha d'obtenir una funció hash basada en l'algorisme *sha256*, el qual retorna un objecte de tipus *string* de 256 bits. Aquest és més fiable que el mencionat anteriorment *MD5* i d'aquesta forma es genera el valor de *vx*. Per generar-lo s'utilitza la llibreria Hashlib[7] de Python.
A la clau *vx* se li ha d'aplicar l'algoritme AEk que és un esquema "Authenticated encryption"[9] amb clau k. Aquest algorisme consisteix en aplicar un AES sobre *vx* i després concatenar a aquest resultat el resultat del hash d'ell mateix.

```
class AESCipher(object):

    def __init__(self, key):
        self.bs = 16
        self.key = key

    def encrypt(self, raw):
```

```
        raw = self._pad(raw)
        #actually is random string with 16 bytes length
        iv = "LSFUw7ndObHjY2bA"
        cipher = AES.new(self.key, AES.MODE_CBC, iv)
        return base64.b64encode(cipher.encrypt(raw))

    def decrypt(self, enc):
        enc = base64.b64decode(enc)
        iv = enc[:AES.block_size]
        cipher = AES.new(self.key, AES.MODE_CBC, iv)
        return self._unpad(cipher
                .decrypt(enc[AES.block_size:]))
                .decode('utf-8')

    def _pad(self, s):
        return s + (self.bs - len(s) % self.bs) *
                chr(self.bs - len(s) % self.bs)

    @staticmethod
    def _unpad(s):
        return s[:-ord(s[len(s)-1:])]
```

En termes de programació, consisteix en crear un object AESCipher amb clau k i sobre aquest objecte utilitzar la funció *encrypt* passant la variable x concatenada amb vx.
Sobre el resultat anterior cal aplicar l'algorisme hmac i concatenar el resultat.
AESCipher és un objecte creat per poder encriptar i desencriptar en AES amb facilitat amb l'ajuda de la llibreria PyCrypto[10] que té implementat un bon sistema AES. També implementa molts altres sistemes criptogràfics com DES, RSA o ElGamal entre altres.

```
def AEk(k,vx,x):

    aes = AESCipher(str(k))
    encrypted_vx = aes.encrypt(str(x)+vx)
    new_vx = hmac.new(k)
```

```
    new_vx.update(encrypted_vx)
    new_vx = encrypted_vx + new_vx.hexdigest()

    return new_vx
```

En aquesta part del sistema ja es posseeixen totes les dades de la iteració i queden preparades per ser estructurades i enviades al servidor.
Pel que fa a l'estructura de dades, s'ha decidit utilitzar la llibreria JSON, creant un objecte el qual conté les dades que el client vol emmagatzemar i una llista d'objectes, amb forma de tupla, creades a cada una de les voltes del bucle les quals contenen tant l'índex de la iteració com la clau encriptada, resultat de l'algorisme de xifrat de AEk.

```
def prepare_data_to_send(dataBlock,x,new_vx):

    new_element = element()
    new_element.i = x
    new_element.vi = new_vx
    dataBlock.token_array.append(new_element.toJson())
```

Just abans d'enviar les dades al servidor l'últim pas que ha de fer el client és emmagatzemar les seves claus i la configuarció que ha utilitzat en un fitxer que no compartirà.

Una vegada el servidor ha rebut les dades li envia al client un missatge de confirmació. És en aquest moment quan es pot donar per tancada la fase de configuració, la primera fase del sistema PDP.

4.2 Fase de verificació

Per validar que el servidor posseeix les dades del client, aquest sol·licita una prova de la possessió de dades al servidor. Per dur a terme la prova de possessió de dades el client escull un índex del token que servirà per reptar al servidor. L'algorisme del client accedeix a les metadades que s'han guardat a la fase de set up i s'obté el token i la configuració en la que es van obtenir aquestes dades, que ha d'utilitzar per validar les dades que envia el servidor.
Una vegada obtingudes les dades, el client llença el "Challenge" al servidor, on se

li envia l'índex que vol validar el client juntament amb les dues claus generades a partir de la PRF tal com es va fer a l'algorisme de configuració, ki i ci. Amb totes aquestes dades el servidor ha de calcular les evidències que ha de validar el client. Aquestes evidencies consisteixen en calcular, com en la fase anterior, el hash de la permutació de les dades (PDP). De fet si el client ha passat les dades de manera correcta hauria de ser exactament el mateix procediment que ha fet el client en la fase de set up per aquell index concret i el servidor no ha modificat cap dels blocs de dades que s'utilitzen per la creació del HASH.
Amb la funció hash calculada solament falta enviar els resultars al client el qual validarà si el servidor conserva les dades intactes. El servidor, per tant, enviarà al client tant el càlcul del hash com el token que el client li va enviar en la fase anterior.
Una vegada el client obté resposta, procedeix a fer l'última validació de la demostració de possessió de dades.

```
json_checker = json.loads(check_data)
z = json_checker["z"]
vi = json_checker["vi"]
k = str(json_data["k"])

checker = AEk(k,z,i)

if str(checker) == str(vi):
    print "DATA_VERIFIED!"
else:
    print
    print "DATA_SERVER_HAS_DELETED_OR_MODIFIED"
```

Sobre el hash que ha calculat el server s'ha d'aplicar l'algoritme AEk, del qual hem parlat a la fase anterior, amb la clau k ,que el client té emmagatzemada al fitxer de metadata. Si el resultat de l'algorisme és igual al token que el servidor tenia emmagatzemat, el client pot deduir que el servidor conserva les dades.

4.3 Comunicació client-servidor

Aquest sistema està pensat per un entorn client-servidor en anglés *client-server*. La forma que existeix per treballar amb una comunicació *client-server* és amb

la llibreria de sockets[11], de Pthon. Durant la implementació d'aquest sistema s'han utilitzat els sockets INET i del tipus STREAM.
Podem dir que un *socket* és un parell de tuples formades per una adreça IP i un número port que serveixen per comunicar diferents programes que es poden trobar en màquines diferents. A més, els *sockets* poden garantir la transmisió de tots els objectes sense errors ni omissions i que aquests objectes es rebran ordenats tal i com s'han enviat.
Que un socket sigui INET solament vol dir que els programes operen en diferents sistemes units mitjançant una xarxa TCP/IP. El tipus més importants dels sockets INET són els de tipus Stream, els quals utilitzen el protocol TCP. Aquest proporciona un fluxe de dades bidireccional, seqüencial, sense duplicació de paquets i lliure d'errors.

Primer, una vegada les dades dels algorismes del sistema han estat tractades i preparades per enviar des del client fins al servidor, s'ha de crear un socket i establir la connexió amb el socket que ja està creat al servidor esperant la petició del client.

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_address = (ip, port)
sock.connect(server_address)
```

Una vegada s'ha establert la comunicació entre el client i el servidor, el client li envia tot el paquet de dades en format de string.

```
sock.sendall(message)
```

Llavors espera la resposta del servidor, comprovant que tot ha funcionat correctament. En aquest cas s'està enviant un simple missatge de confirmació o d'error des del servidor al client.

```
try:
    stored_data_file.write(data)
        connection.sendall("all data receeived and saved")
except ValueError as err:
    connection.sendall("error saving data! Try it later!")
```

Una vegada el client rep la confirmació s'acaba l'execució, en el cas del primer algorisme (fase de confirmació), o continua amb l'execució de la segona part de la fase de verificació.

5 Resultats experimentals

5.1 Estudi temporal

En aquest apartat es realitza un estudi temporal amb diferents configuracions i entrades de dades. L'objectiu d'aquest projecte és provar que el sistema funciona i que és un sistema PDP viable.
Les proves s'han realitzat des d'un ordinador portàtil amb un procesador intel i7 i 8 GB de RAM i amb Sistema Operatiu Ubuntu versió 14.0.

Per poder fer una bateria de proves suficient àmplia per provar que els algorismes són eficients i viables, principalment es treballa amb tres mides de fitxers i amb quatre configuracions diferents. Amb aquests tres fitxers es pot veure com afecta la mida del fitxer al sistema, i amb les quatre configuracions per sistema es pot validar com afecta la mida de les metadades al comportament dels algorismes.

Primer és fan les proves amb el fitxer més petit i, una vegada es comprova aquest, es continua pels de mida superior. Per part de la configuració, hi ha 3 elements modificables en aquest sistema: les vegades que l'usuari pot reptar al servidor(Challenges), el numero de blocs necesaris per generar el hash de confirmació(r) i, per últim, el numero de blocs en que es divideixen les dades(Blocs).

D'aquesta manera ja es possible fer-se una idea del temps que li pren al sistema realitzar diferentes configuracions amb diferents fitxers, i així obtenir les bases de l'estudi.

Primer fitxer La seva mida és de 7'6 Kb.
Per aquest fitxer les quatre configuracions són:

Primera configuració

- 100 challenges
- r: 64
- Blocs: 128

Segona configuració

- 6000 challenges
- r: 64

- Blocs: 128

Tercera configuració

- 100 challenges
- r: 256
- Blocs: 1024

Quarta configuració

- 6000 challenges
- r: 256
- Blocs: 1024

Els temps obtinguts a les quatre configuracions a la fase de configuració són: a la primera, 23'634 ms; la segona, 1189'37 ms; la tercera, 63'75 ms i la quarta 3527'517 ms.

Per la fase de verificació els temps obtinguts són: la primera, 6'038 ms; la segona, 24'83 ms; la tercera, 7'369 ms i la quarta, de 21'658 ms.

Amb el primer fitxer es pot obtenir una de les claus de l'algorisme. S'observa que la fase de verificació, pràcticament, no varia el temps d'execució entre escollir una generació de claus de 64 blocs de dades o de 256, per tant, es pot afirmar que el cost de l'algorisme PRP, el que s'utilitza per la generació de claus, és minim comparat amb el cost de cercar les dades al JSON.

Segon fitxer La seva mida és de 379'3 Kb.
Per aquest fitxer les quatre configuracions són:

Primera configuració

- 100 challenges
- r: 64
- Blocs: 128

Segona configuració

- 4000 challenges
- r: 64
- Blocs: 128

Tercera configuració

- 100 challenges
- r: 512
- Blocs: 2048

Quarta configuració

- 4000 challenges
- r: 512
- Blocs: 2048

Els temps obtinguts a les quatre configuracions a la fase de configuració són: a la primera, 173'070 ms; la segona, 2794'247 ms; la tercera, 269'214 ms i la quarta 5709'892 ms.

Per la fase de verificació els temps obtinguts són: la primera, 155'751 ms; la segona, 166'751 ms; la tercera, 157'767 ms i la quarta, de 172'698 ms.

Tercer fitxer La seva mida és de 838'5 Kb.
Per aquest fitxer les quatre configuracions són:

Primera configuració

- 100 challenges
- r: 64
- Blocs: 128

Segona configuració

- 3000 challenges
- r: 64
- Blocs: 128

Tercera configuració

- 100 challenges
- r: 512
- Blocs: 2048

Quarta configuració

- 3000 challenges
- r: 512
- Blocs: 2048

En el tercer fitxers els temps obtinguts a les quatre configuracions a la fase de configuració són: la primera tarda 433'832 ms, la segona tarda 4065'127 ms, la tercera tarda 420'398 ms i la quarta 5238'270 ms.

Per la fase de verificació els temps obtinguts són: la primera, 352'781 ms; la segona, 350'76 ms; la tercera, 360'445 ms i la quarta tarda un temps de 369'015 ms.

Amb el fitxer de major mida ja es pot començar a veure que el cost de la fase de configuració creix segons els reptes que se li voldrà demanar al servidor.

Per contra, la fase de verificació no variarà tant, tindrà un cost pràcticament lineal o constant.
Una vegada calculat un mostreig ampli de dades es pot començar a intuir el comportament que tindrà sistema amb diferents tipus de dades. Les conclusions que es poden extreure d'aquest primer mostreig són que, primer, el que més afectarà a la velocitat a la fase de validació és la mida.
Segon, el que més afecta a la fase de configuració és el nombre de reptes que es volen realitzar i el nombre de blocs en que es divideixen les dades.

A la segona part de l'estudi s'intentarà obtenir una conclusió dels diferents tipus d'execucions que es podrien obtenir en un sistema real. Per exemple, mirar exactament com afecta la mida del fitxer a una configuració concreta, o obtenir la diferencia temporal de l'increment de reptes desitjats, entre altres proves.
Aquestes segones proves es realitzen variant les configuracions amb cinc valors diferents, i així crear un gràfic, per obtenir una prova més visual del comportament del sistema.

En el primer estudi, ens interessa conèixer com afecta la mida de les dades al sistema. Per aquest experiment utilitzarem una configuració bastant senzilla. En aquest cas una que realitza 200 *challenges*, amb pocs blocs de dades(128) i pocs blocs també per generar el token(64).
D'aquesta manera es pot assegurar que el temps que li pren al sistema realitzar les dues fases és purament degut a la mida de les dades. La gràfica generada s'ha creat a partir de quatre fitxers. Ja es pot veure la tendència que pren el temps depenent de la mida del fitxer.

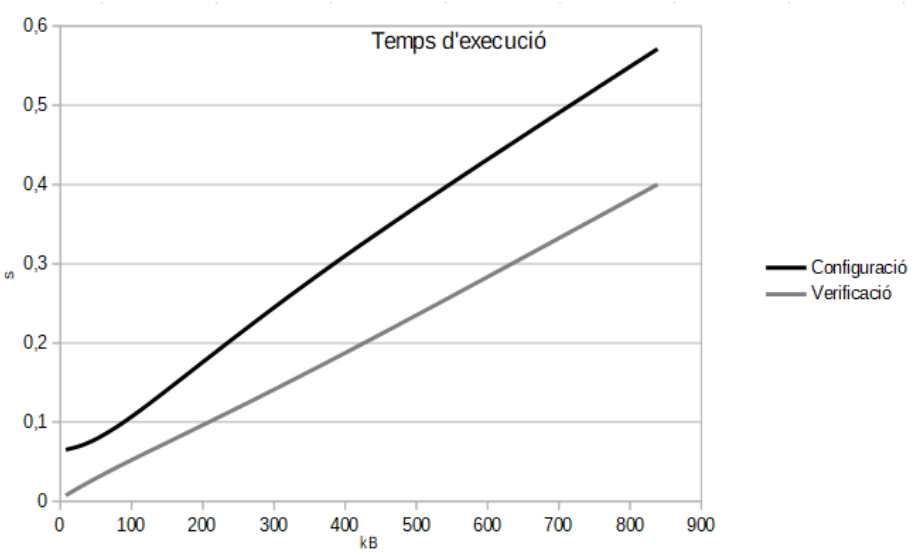

Figura 8: Gràfic temporal depenent de la mida del fitxer

Com es pot observar a la gràfica 8, el temps de la fase de verificació és completament lineal, ja que l'única tasca que realitza aquesta fase és trobar a un JSON la clau reclamada y càlcular el token. Com sempre són 100 reptes podem assegurar que el temps és el mateix en els tres casos. Per tant podem veure que el cost de calcular la permutació PRP és lineal en relació a la mida de les dades. El temps d'execució també és lineal, del que podem deduir que la mida del fitxer que volem emmagatzemar, no afecta tampoc als algorismes ni de PRP ni de PRF per la part del client. Simplement, pesa molt la tasca de gestió de dades per dur a terme la comunicació entre el client i el servidor.

També volem observar més concretament com estan relacionats el nombre

de reptes que es volen realitzar al servidor amb el temps que es tarda en generar, tractar i transmetre les dades. Per aquest estudi s'utilitzarà un fitxer de 100 kB, que és un fitxer amb una mida suficient per poder dir que el tractament de dades no es negligible, ja que com hem vist anteriorment, per tractar 200 *challenges* tarda més de 100ms.

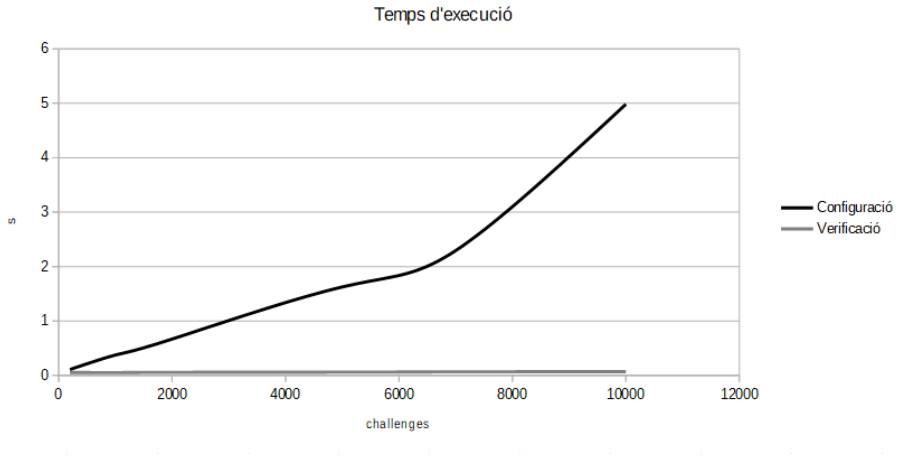

Figura 9: Gràfic temporal depenent dels reptes que es volen realitzar

Com es pot observar a la gràfica 9, quan s'incrementen els reptes que s'han de fer al servidor es nota la diferencia a la fase de configuració, però el cost és constant a la fase de verificació.
Donat que la mida de les dades que es transporten és molt més gran, el cost que representa la creació del JSON i el transport de dades comencen a ser una part important del cost temporal del sistema.
Com es pot deduir, el bucle encarregat de generar tots els tokens per dur a terme els reptes representa un cost directament proporcional als reptes que es volen, i cada una d'aquestes iteracions tarda exactament el mateix temps. Per tant d'aquesta part del sistema coneixem que es lineal.
El que es pot concloure observant aquesta gràfica és que a la fase de configuració a partir de certs reptes el transport comença a pesar, temporalment parlant, respecte al bucle principal del sistema i incrementa el temps, però a la fase de validació el numero de reptes no influeix en res, és un cost de O(1).

Per concluir la bateria de proves, veurem com influencia la seguretat en el temps.

En aquestes s'utilitzara el fitxer de 100kB també ja que com abans s'explicava és un fitxer amb una mida suficient per realitzar proves amb 2000 reptes.

El que canviarà a cada una de les execucions són el nombre de blocs en que es dividiran les dades. Per generar els *tokens* de validació s'utilitzarà la meitat de blocs en que s'han dividit les dades.

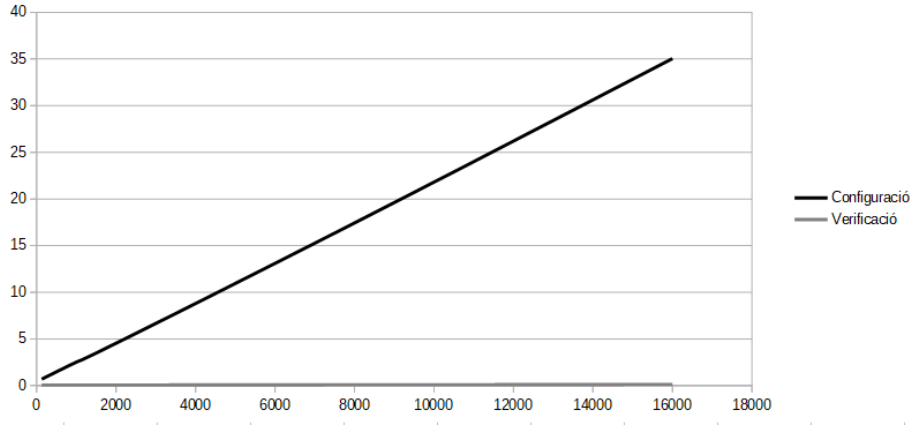

Figura 10: Gràfic temporal depenent dels blocs per validació

De la gràfica 10 es pot deduir que el cost és lineal, però a la validació pràcticament no varia el temps. A contiunuació es presenta una taula del temps que ha costat fer la fase de verificació, ja que de la gràfica no es pot veure la diferencia entre totes les configuracions ja que els canvis són minims comparats amb el temps de configuració.

Blocs	T.configuració	T. verificació
128/64	0.6929s	0.05322s
1024/512	2.5638s	0.05556s
2048/1024	4.6453s	0.06093s
8000/4000	17.4356s	0.08293s
16000/8000	35.0479s	0.1189s

Taula 1: Taula de temps amb diferentes quantitats de blocs

Es pot observar a la taula 1 que el temps de verificació incrementa, però d'una forma molt més suau que la configuració, ja que l'increment a la configuració es multiplica pel número de reptes a part del nombre de blocs per validació.

Una vegada fet l'estudi temporal i deduït que pràcticament el temps és lineal

depenent de les configuracions s'intentarà fer una prova amb un fitxer el més real posible, en aquest cas s'ha agafat un llibre i es calcularà el que tarda el sistema.

Per un llibre de 5,5Mb i 40000 reptes al servidor, amb 900 blocs per verificació el temps que li pren al sistema fer la fase de configuració és d'una mica més de 7 minuts i 30 segons.
En canvi, al fer la fase de verificació el sistema tarda aproximadament 2 segons. En aquest cas, amb el fitxer tant gran, es pot observar la gran diferència que hi ha entre les dues fases. Com és normal ja que a la fase de configuració per cada repte es repeteix el que es realitza a la fase de verificació a més de dur a terme el transport.

5.2 Estudi probabilitat

A l'article "Scalable and eficient Provable data possession" fan també un estudi de la probabilitat que té el servidor de donar una resposta correcta sense posseir tots els blocs de dades o tenint blocs de dades modificats. A l'estudi, els autors asseguren que la probabilitat de que cada repte que fa el client no detecti un canvi és:

$$P = (1 - \frac{m}{d})^r \tag{1}$$

On **P** és la probabilitat, **m** el nombre de blocs, **d** els blocs modificats i **r** el nombre de blocs utilitzats per generar el token.
És possible que s'hagi modificat un bloc de dades i no es detecti el canvi, si es dóna la casualitat que en aquella verificació no s'ha utilitzat el mateix bloc per crear el hash que actua de validador.

Quina probabilitat hi ha de que després de 100 reptes el client no detecti que les seves dades no han estat modificades o parcialment borrades?

Seria molt improbable, ja que la formula anterior és per un sol repte. Si calculem que passi **n** cops seria:

$$P = ((1 - \frac{m}{d})^r)^n \tag{2}$$

Per exemple, suposem una configuració on dividim les dades en 100 blocs, i es necesiten 64 blocs per validació amb 100 reptes al servidor. Suposant que

solament s'ha modificat un bloc de dades la fórmula és la següent:

$$P = ((1 - \frac{1}{100})^{128})^{100} \tag{3}$$

El percentatge d'error d'un repte és:

$$P = 0'27 = 27\% \tag{4}$$

Amb el total de reptes, el percentatge és infinitament més petit:

$$P = 1.35x10^{-56} \tag{5}$$

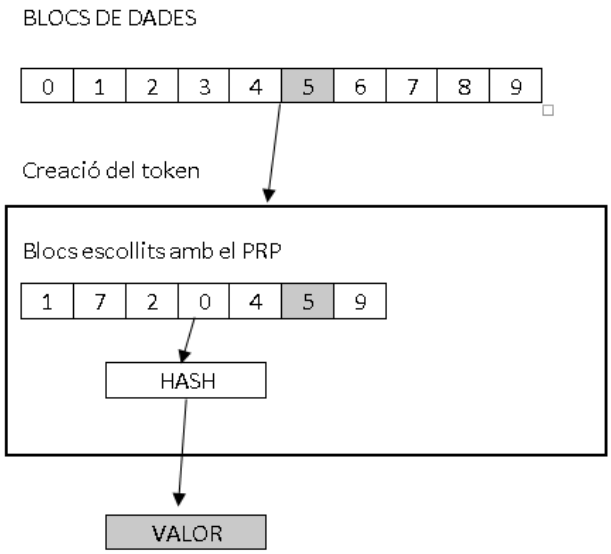

Figura 11: Esquema on es detecta l'error

Com veiem a la imatge 11, a l'algorisme per obtenir la clau s'ha utilitzat un bloc de dades diferent del que s'havia utilitzat amb el client, per tant aquest a l'hora de validar veurà que no ha donat el mateix resultat.

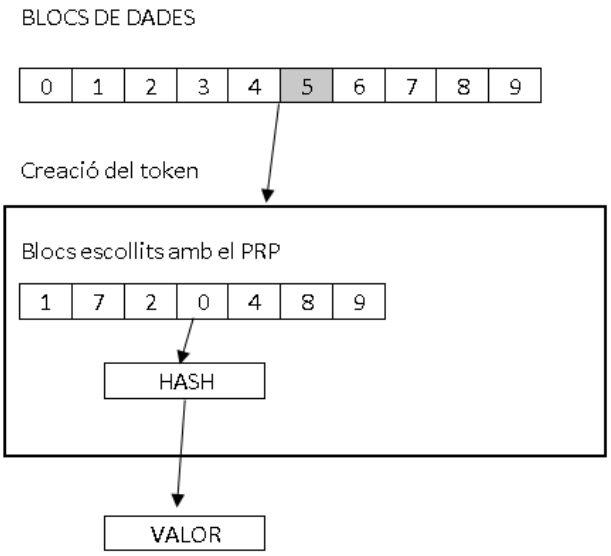

Figura 12: Esquema on no es detecta cap error

En aquest cas el servidor ha tingut la sort de no modificar o esborrar cap dels blocs de dades que s'utilitzen per la generació de claus. Per tant el client en aquest repte veurà que la validació és correcta i no se n'adornarà que s'han esborrat o modificat dades.

6 Conclusions

Per concloure l'estudi, podem assegurar que l'algorisme PDP és un bon sistema de validació de dades remotes en un servidor extern. En termes de seguretat hem pogut comprovar que no és gens senzill poder falsificar les dades d'un sistema d'aquests tipus. Per altra banda, hem vist que el sistema proposat a l'article de G.Ateniese, R. Di Pietro, L. Mancini and G. Tsudik *"Scalable and Efficient Provable Data Possession"*[4] és un sistema més que viable per procedir a aquestes validacions contra el servidor, ja que no solament millora la seguretat d'articles anteriors sinó que els combina per fer un sistema encara més eficient en termes de transport de dades i treball de computació tant a la part del client com del servidor.

De l'estudi hem pogut deduir que en sistemes mitjanament grans amb una màquina no gaire potent, la càrrega temporal no ha sigut excessivament gran, i que amb una potencia de càlcul com la que es pot trobar en els ordinadors d'avui en dia es poden extreure resultats molt positius.

7 Treball futur

A continuació exposo detalladament les parts del projecte per on es podria continuar desenvolupant en un futur. Per una banda, el sistema proposat a l'article de Ateniese et al. anomenat "Scalable and Efficient Provable Data Possession"[4] també proposen mecanisme per quan hi ha qualsevol tipus de canvis a les dades: tant afegir blocs, com borrar-los o modificar-los. Segons la meva implementació, com hem vist, per diferenciar la crida al servidor per emmagatzemar dades o per fer una de les comprovacions s'utilitza el "mode". És un argument d'entrada que serveix per diferenciar l'algorisme que s'ha de cridar tant al client com al servidor. Actualment, aquest camp (mode) pot tenir el valor "store", per l'algorisme d'emmagatzematge, o "challenge", per l'algorisme de repte. En el futur, amb l'incorporació de la possibilitat d'actualitzar un fitxer, el camp mode podria pendre el valor "update", i es podria completar l'entrada de dades que requereix l'algorisme.

Per altra banda, actualment la implementació no disposa d'una interficie gràfica per al client. Per ampliar aquest projecte es podria desenvolupar una sèrie de pantalles en un entorn web. Per fer més usable el programa, es podrien incorporar els camps necessaris per cada un dels modes i una nova forma de mostrar si el servidor ha passat la prova. Personalment recomano utilitzar Angular per desenvolupar aquesta web, ja que és una tecnologia molt completa, amb una gran comunitat i que ofereix moltes possibilitats per treballar entre pantalles i el mateix programa en Python.

Referències

[1] Giuseppe Ateniese , Randal Burns , Reza Curtmola , Joseph Herring , Lea Kissner , Zachary Peterson , Dawn Song, *Provable data possession at untrusted stores*, Proceedings of the 14th ACM conference on Computer and communications security, November 02-October 31, 2007, Alexandria, Virginia, USA
`https://dl.acm.org/citation.cfm?id=1315318`

[2] Ari Juels i Burton S. Kaliski, *PORs: Proofs of retriability for large files*, Proceedings of the 14th ACM conference on Computer and communications security, November 02-October 31, 2007, Alexandria, Virginia, USA
`https://dl.acm.org/citation.cfm?id=1315318`

[3] Funció Hash, (sense data). Wikipedia, Recuperat al 2018
`https://en.wikipedia.org/wiki/Hash_function`

[4] Giuseppe Ateniese , Roberto Di Pietro , Luigi V. Mancini , Gene Tsudik, *Scalable and efficient provable data possession*, Proceedings of the 4th international conference on Security and privacy in communication netowrks, September 22-25, 2008, Istanbul, Turkey
`https://dl.acm.org/citation.cfm?id=1460889`

[5] Advanced Encryption Standard, (sense data). Wikipedia, Recuperat al 2018
`https://en.wikipedia.org/wiki/Advanced_Encryption_Standard`

[6] Validació HMAC, Python api, Python Software Foundation.
`https://docs.python.org/2/library/hmac.html`

[7] Llibreria hashlib, Python api, Python Software Foundation.
`https://docs.python.org/2/library/hashlib.html`

[8] MD5, (sense data). Wikipedia, Recuperat al 2018
`https://es.wikipedia.org/wiki/MD5`

[9] Authenticated encryption, (sense data). Wikipedia, Recuperat al 2018
`https://en.wikipedia.org/wiki/Authenticated_encryption`

[10] pycripto 2.6.1. , Llibreria Python, Dwayne C. Litzenberger
`https://pypi.python.org/pypi/pycrypto`

[11] Llibreria sockets, Python api an wiki, Python Software Foundation.
https://wiki.python.org/moin/HowTo/Sockets